CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Facultad de Informática

TRABAJO FIN DE GRADO

Generación Procedimental de Contenidos en Videojuegos

Autor: Álvaro Gutiérrez Lorenzo

Director: José María Peña Sánchez

Codirector: Luis Peña Sánchez

MADRID, ENERO DE 2014

Index

Summary

I - Español

Este proyecto consiste en el desarrollo de un sistema completo de generación procedimental de misiones para videojuegos.

Buscamos crear, mediante un encadenamiento de algoritmos y un modelado del juego y sus componentes, secuencias de acciones y eventos de juego encadenados entre sí de forma lógica. La realización de estas secuencias de acciones lleva progresivamente hacia un objetivo final. Estas secuencias se conocen en el mundo de los juegos como *misiones*.

Las dos fases principales del proceso son la generación de una misión a partir de un estado de juego inicial y la búsqueda de una misión óptima utilizando ciertos criterios que pueden estar ligados a las propiedades del jugador, dando lugar a misiones adaptativas.

El proyecto contempla el desarrollo íntegro del sistema, lo que incluye tanto el sistema de generación y búsqueda como un videojuego donde integrar el resto del sistema para completarlo. El resultado final es plenamente funcional y jugable.

La base teórica del proyecto proviene de la simbiosis de dos artes: la generación procedimental de contenido y la narración interactiva.

II - English

This project involves the development of a complete procedural game quest generation system.

We seek to build, by linking a series of algorithms, game and game component models, sequences of logically chained game actions and events. The ordered accomplishment of these sequences lead progressively to the fulfillment of a final objective. These sequences are known as *quests* in the videogame world.

The two main parts of the process are quest generation from an initial game state and optimal quest search. This last is achieved by using certain criteria that can defined by the player properties, thus giving birth to adaptive quests.

In this project. The system is comprehensively developed, including the quest generation and optimal search, as well as a full videogame, in which the rest of the system will be embedded so as to complete it. The final result is fully functional and playable.

The theoretical basis of the project comes from the symbiosis of two different arts: procedural content generation and interactive storytelling.

Theoretical basis

1 - Introduction

As happens with most multimedia technologies, upon reaching a technical level that provides the possibility of making anything that could possibly be imagined at the moment two ways of going forward are open (check Intel's *Tick-tock model* [1]).

The first is to make development easier, faster, and cheaper by improving current technologies and techniques, and also by conceiving new ones. This dynamic brings new possibilities such as introducing new platforms to the scene, or opening a door to unexpected new technological tiers.

The second path is to expand the range of uses that exists based on the current technology. This means exploring new ways to exploit the present situation, a search for originality that carries breakthroughs in the manners of interacting and experiencing that multimedia technology.

This project walks that second path, aiming to explore new approaches to interact and feel videogames.

Among the fields that have recently started to be researched, the project settles its feet in two different fields and builds a bridge between them. Both fields experience many ongoing researches, and are generally considered promising work grounds in the world of videogames.

One is the **Procedural Content Generation** (PCG). Julian Togelius et al. discuss on their 2011 paper *"What is Procedural Content Generation?"*[2] how PCG, in the context of videogames, should be defined. They eventually build the following definition: "*PCG is the algorithmical creation of game content with limited or indirect user input*". Furthermore, they note the possibility of PCG methods being random and/or adaptive.

"Adaptive" is a concept firmly attached to the second field this project focuses on: **Interactive Storytelling** (IS). As David Thue, Vadim Bulitko and Marcia Spatch state, *"When a story's events are chosen based on feedback from its audience, the telling of that story is **Interactive**"* [3]. In other words, we can speak of a story that evolves adapting to match the audience, generally looking for that audience to like and immerse itself in the story.

While the emergence of PCG is linked to the electronic entertainment arrival, interactive storytelling is an older concept that fits the possibilities and interests of videogames.

As explained before, the project picks technology from both PCG and IS, but the main goal of it is to build a functional system with it, an actual game that will generate quests for the player to solve. In the context of videogames, quests are sequences of

story events and particular objectives meant to be accomplished by the player, who is able to make decisions on how to accomplish (or fail) the objectives. The game system collects information about the player and the consequences of his/her decisions to adapt further generated quests. That way, each gameplay will result in a different story, influenced by the player, who is actually the audience and protagonist of that story.

2 - About procedural content generation

Procedural content generation (PCG) in videogames is the process of creating game material (scenarios, game quests, dialogues, puzzles… anything that will appear or affect the game) by means of automatic or programmatic techniques. The process may occur at any time:

- during the development in order to produce material that will be included in the final game (games with huge, open worlds often produce scenarios procedurally then are reviewed, tested and embellished)
- just before the game is going to be played (when a game level loads, items and enemies can be procedurally created and positioned)
- on run/play time (characters dialogues that are generated as the conversation develops for instance)
- etc.

As Julian Togelius et al. explain in *"What is Procedural Content Generation?"*[2], the process is always an algorithm, or algorithm driven process, thus, human content creation is not procedural content generation. Besides, frequently PCG is supposed to possess some properties that actually are not necessarily found in PCG:

- PCG in not necessarily random. In practice, PCG has been used to deterministically produce the same content using a small seed in order to save memory.
- PCG in not necessarily adaptive. The process may or may not take into account emergent factors to generate content: PCG can be totally unaware of the gameplay results when creating new content.

In this particular project, generated context is random, and also is intended to be adaptive (see next section and section 10 for more information about the adaptive components of the project).

Nowadays, PCG in games experiences a popularity growth, every year dozens of games that feature some kind of procedural content generation enter the market. The most common form of PCG on games currently is map and level design, but other types of content have been procedurally generated: items and weapons, terrain props

and population, enemies, names, character properties, or quests, as we do in this project.

The main benefit for the player that PCG brings is unlimited possibilities of playthroughs, as the game would present different levels, scenarios, quests … every time it is executed. For developers, PCG can be a useful tool and time saver. Huge amounts of content can be generated using PCG techniques, and has made possible many games that would have been radically different, or impossible to develop without PCG.

3 - About interactive storytelling

The art of storytelling encompasses a wide set of different archetypes. A first line could be drawn between an interactive and non-interactive way of telling a story. As quoted in the previous chapter, the guideline D. Thue, V. Bulitko and M. Spatch (from the University of Alberta) use to make the difference reads "*When a story's events are chosen based on feedback from its audience, the telling of that story is* ***Interactive***"[3].

Using that guideline, the most common ways of transmitting stories would be tagged as non-interactive. Usual books, films, plays lack interactivity, the same story is being told and spread. The book had written the same sequence of events whoever it opened and read it. That was, nevertheless, one of the ideas Gutenberg had in mind when he started printing books in 1450, the idea of communicating an unchanged story.

On the other side, interactive storytellers transform the story using the information they get from the audience. Traditionally, a storyteller that would like to adapt the story for its public could make the events happen in the same place the audience comes from, change the names and references with other closer to the public context. Even more interesting than punctual changes, the storyteller would be able to change the order, or the magnitude of the story events according to the people's reactions and emotions to previous events.

Depending on the context and the story, both categories bring different benefits and problems that make either suitable for each case. For the particular case of videogames, the concept of interactive story quickly made its place, so much so that today it is commonly supposed that if a videogame encloses a story, the player will be able to influence, even if minimally, the progress of the story.

Moreover, the genre variety in videogames results in a variety of ways of presenting a story. Federico Peinado (Universidad Complutense de Madrid), David Thue, Vadim Bulitko and Marcia Spatch (University of Alberta) agree in classifying how

stories are told in videogames in an interactivity degree scale, ranging from *predefined script* to *emerging plot*[4].

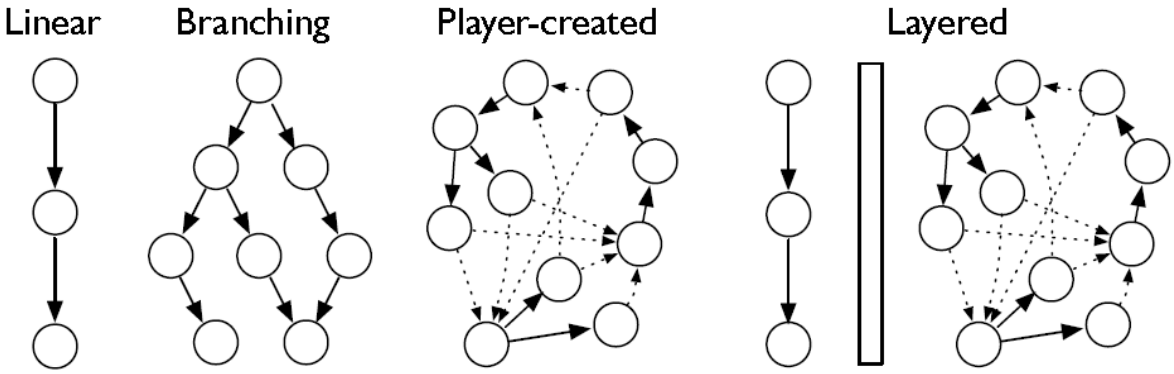

Figure 1: Different storytelling schemes on videogames, extracted from the PaSSAGE presentation [3]

Possibilities brought by the videogame format such as the interactivity, multisensory resources, narrative power as well as every possibility derived from the software essence of the videogame have expanded the interactive storytelling horizons. Nevertheless, videogames benefit too from this symbiosis.

Among the desirable properties interactive storytelling provides to videogames using it can be found [3] [5]:

- Player agency, or *player influence in the story*, that results in a feeling of freedom inside the game.
- Variety, each gameplay will be different, thus adding replay value to the game and expanding the game itself.
- Immersion, by bringing the game closer to the player's context and adapting to match his/her personality, reducing the distance between the player and the game.
- Adaptability, the game molds itself to the player gaming characteristics, triggering more events that are presumably more entertaining for each particular player profile, and maybe hiding the ones that could bore that profile.

Player profiles are a major key for interactive storytelling in videogames. Players, as the audience of the story, are the input used to modify the story and make it more suitable for them. The story is modified by combining events and parts that are more likely to please the player, but for that the story events and parts need to be evaluated to know if they will match or not the player.

Evaluating each story event against each player is impossible. Still, some methods suggest to simplify the problem by representing the possible different players using a model composed of player profiles.

Robin D. Laws suggests the following set of profiles:

“

- Power Gamer: searches for new abilities and special equipment.

- Butt-kicker: always waits for a chance to fight with someone else.

- Tactician: feels happy in the anticlimax scenes, thinking about logical obstacles to overcome.

- Specialist: just wants to do the things that his/her favorite character does.

- Method Actor: needs situations that test his personality traits.

- Storyteller: looks for complex plot threads and action movement.

- Casual Gamer: remains in the background (and generally has a very low degree of participation).”[6][7]

As an example, a story part (or *encounter*, in PaSSAGE’s terminology [3]) that involves fighting and no dialogues can be considered more suitable for a *Butt-kicker* or *Tactician* player, and boring or disturbing for a *Storyteller* or *Method* Actor one.

With this, different stories would be built for players matching different profiles, aiming to make the best possible experience for that player. Of course, this is only one possible method, and other approaches can be developed. The PaSSAGE team displays the algorithm they apply using this same model to get the best story encounters for a particular player in their presentation [3].

In previous interactive storytelling projects, such as the ones presented by F. Peinado [5], it can be seen that the story is always built by an artificial intelligence (AI) that applies models and algorithms like the one previously mentioned to adapt the story to the audience. It remains a question: ¿which is the role of the game designer?

Some games aim to tell a particular story with minor influence from the player, others have no preconceived story at all. Inside this range, the role of the game designer would be to write pieces of stories ranging from a full, linear story to little, atomic story events. Between those extreme cases, story parts could be big story arches as the ones in a play, or small story pieces like “killed the wolf” or “killed the wolf then reached the forest and found the secret cave that lead to the treasure”. Other tools like the “Author Goals” described by M. Riedl, D. Thue and V. Bulitko [8] can harness a story built up with relatively small parts, thus keeping a high degree of player agency without losing control over the story.

Summarizing this last concepts together, artificial intelligence that combines and adapts human made story parts to create a better, particular player oriented story and experience is what today is known as interactive storytelling in videogames.

Project development: System and implementation

4 - Overview

This section gives a general view of the system and introduce the concepts that will be handled in this report.

In this project we built a complete functional system, composed of several parts. The system builds procedural quests from initial game states called contexts. Built quests are later represented in a video game, so they can be played as in any usual game.

The project encompasses the whole process: context generation, valid quest construction, quest evaluation, search algorithm to seek for the best possible quest and eventually the game that makes quest playable.

The project is extensible, and future steps, such as, introducing a system feedback that would take information from the player and use it as evaluation criteria for the quest search algorithm have not been yet explored but would be possible to be included in the presented architecture.

Here is a conceptual map of the whole system:

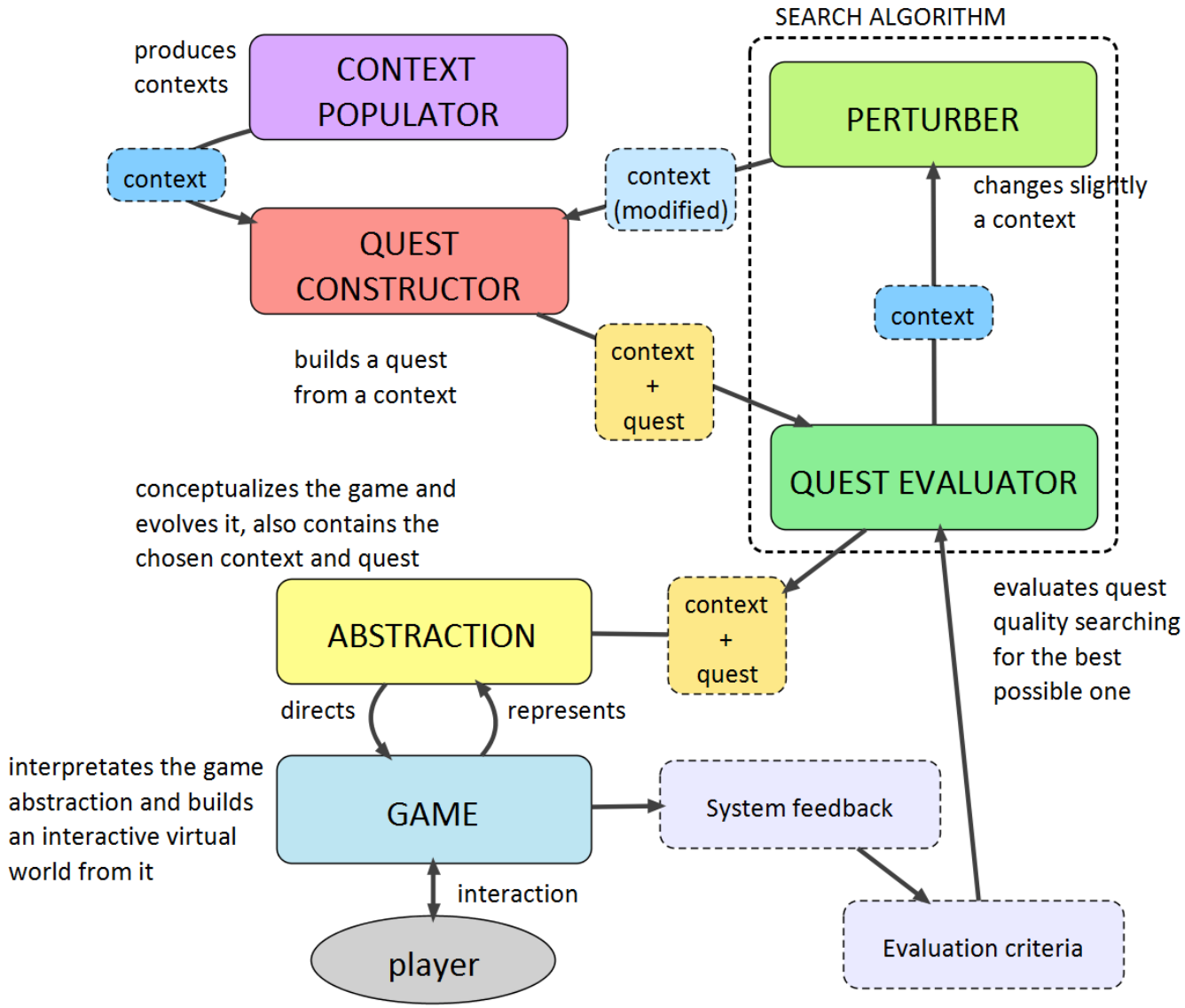

Figure 2: System overview map

Different parts outline:

- Context populator: a process that creates a context and fills randomly its components. A context is a starting point of the game, a complex object that encloses all the game objects (characters, scenarios, items…) and their properties, the game state (in the form of labels, section 5.b), and the events and actions that may occur in the game (in the form of rules, section 5.c).
- Quest construction algorithm: an algorithm that builds a quest from a context. Each context produces only one quest (or none, if it is impossible to be achieved).

- Abstract model (abstraction): the abstract model is the conjunction of label types, label operating rules, rule producing methods, rule, game object types, and logical constraints that make the game work. Contains a single context and its corresponding quest.
- Game: represents and makes interactive the game abstraction. Through the interaction with the game, the player activates context rules and triggers changes to the game state. The aim of the player is to complete the quest.
- Search algorithm: the algorithm that searches stochastically for the best possible quest. The algorithm implements a hybrid heuristic optimization technique. It evaluates each quest by giving it a fitness value. After evaluating a quest, the algorithm changes slightly the context then evaluates the new result. After a given number of iterations, the best quest is kept and transferred to the game abstraction so the game may begin.

5 - Abstract model

a. Context

In order to search for an appropriate quest to deliver, the system needs to create and modify many different initial states of the game. The root object of the model is the game state, and we call it a *context*. A context is the full state of a game at a particular moment, and may be created, randomized, initialized, cloned, reset, read or used as the current context for the game engine.

A context contains:

- Game objects: characters, items, enemies, buildings, locations …
- Labels: known information, accomplished partial objectives, event outcomes …
- Rules: game actions of events that, under certain preconditions, change the game state. e.g.: "*if path is not blocked, Jack enters the castle*", "*if the shopkeeper is alive, and player has at least 100 coins, player buys the key and loses 100 coins*"

Game objects properties are chosen according to the game needs. Check section 6.b to see game objects included in the game we developed for this project. In the following parts the concepts of *label* and *rule* will explained in depth.

Lots of context objects are generated during the search process; many are kept temporary to produce clones which are modified to seek for better results, many are discarded and the best result is eventually used by the game to load the quest. Check section 8 to see how context objects are handled during the search process.

b. Label

Labels are the atomic object of the model. They represent a piece of information of the game state, a basic concept such as: "Player is in the castle", "Player has 100 coins", "Character 2 has the blue key", "Jack is dead", "The chicken is in the box"…

Statements labels represent open and extensible information that changes dynamically along the game, static data is best stored as game object properties. For instance, if in the game characters always stay in the same scenario, that scenario should be stored as a property of each character. On the other hand, if characters may move around the map, their location has to be recorded with labels. It is possible to keep static information as labels too, but it is advisable to make the difference and save label types only for dynamic data.

In practice label is a structure that contains:

- Label type: this is the concept the label represents (e.g.: "Player has", "Character knows", "Item belongs to"…).
- One or more parameters: in this project, two integer parameters showed to be enough for all needs, other games may need otherwise. Parameters are used as needed, and mean different things according to the label type. e.g.: ["Player is in" , 1] , here "1" is the scenario id ; ["Jack has" 32, 220] , here 32 could be the item id 32 (maybe something like "coin") and 220 the quantity of that kind of item Jack possesses. Parameter types can be different, we chose integer for simplicity when operating with labels.
- Negation state: a label may be negative. When used as conditions, a negative label implies that the label is not present among the collection of labels that must verify the condition. This property is critical when operating with labels, see next paragraphs.

Also, when adding a negative label into an existing collection of labels, if the label is already present in that collection, it will be removed, if it is not, for some cases (mainly labels kept as conditions) the negative label will be included as is into the collection, and for the rest of the cases it will not.

Frequently, labels are added to other labels, or to label collections. According to their meaning, adding two labels produce different results. Before adding labels, a compatibility function must be defined, as only two compatible labels can be added. In our implementation we found two different compatibility criteria:

- Labels are the same type (used for the vast majority of the label type cases)
- Labels are the same type and their first parameter is the same (we used this criterion for item stacking labels, as only items of the same type can be added, for instance: "Has 100 coins" and "Has 2 coins" are compatible an can be added

to produce "Has 102 coins", but "Has 100 coins" and "Has 2 swords" are not compatible, even if they share the same type".

Once two labels are verified to be compatible, they can be added. As said, how the result is calculated depends on the meaning of the label type, and will be different for each game. Here are the cases we found in developing our implementation, please take them only as examples of practical situations:

- Many label types are not possible to be added: e.g.: "Player is in the castle"
- Label_A + Label_B produces a new label by adding a parameter an keeping the type and possibly other parameters. e.g.: "Has 1 coin" + "Has 10 coins" = "Has 11 coins"
- Label_A + ¬ Label_A = Ø. Usually, adding a label to its negative equivalent means removing both labels producing a void result. e.g.: "Player is in the house" + "Player is NOT in the house" = Ø.
- A particular case occurs with items: "Has 50 coins" + "Has NOT 20 coins" = "Has 30 coins".
- Also, "Has 5 coins." + "Has NOT 15 coins" may produce "Has NOT 10 coins" or Ø. The first option is needed when labels are treated as conditions.

Notice the two different cases: adding labels and adding labels *as conditions*. In general terms, when labels are treated as conditions, the system wants to keep track of the lacking labels (*negative labels*), whereas when labels are plainly added, negative labels are not kept.

Adding a label (e.g., label A) to a label collection is done by checking compatibility between the label to be added and each label of the collection until a compatible label is found. When a compatible label is found, the original label is removed from the collection. Both labels are added and the result is introduced into the collection. If no compatible labels are found inside the collection:

- if label A is positive, it is added to the collection
- if label A is negative, two possibilities:
    - if labels are conditions, label A is added to the collection
    - otherwise, label A is discarded

Adding two label collections is done just by picking each label from one of the collections and adding it to the other.

c. Rule

A *rule* is the conceptualization of an event or action in the game. Rules modify the game state by adding, removing or modifying labels of the context. Frequently, rules are action the player has done or events he/she has triggered by performing an action.

Here is an example of a rule: "Player trades a rubber chicken for a blue key with the shopkeeper". This rule changes the game state by removing the "rubber chicken"

from the inventory of the player and adding the "blue key". These changes are what we call *postconditions*, and are labels (the same object seen on the previous section).

Let's keep up with the example. So far we have identified two postcondition labels, "Player has blue key." And "Player has NOT rubber chicken". Rules have *preconditions* too, a different set of labels that must be contained in the game state for the rule to be applied. The shopkeeper will not give out her key if the player does not give her the rubber chicken, and for that player must first *have* the rubber chicken. That is the label "Player has rubber chicken". Other preconditions that can be outlined are: "Player is in the same scenario than the shopkeeper" "The shopkeeper is alive" "Shopkeeper is not busy" "Shopkeeper does NOT hate the player" … It fully depends on the game needs and mechanics, if the shopkeeper may not die, the rule does not need the check she is alive, if player may get the key only if the blue door has been discovered, rule needs to verify it, and so on.

Rules are added to the context when first populating it. A game could take rules from a library of predefined rules, but in this project we create and modify them randomly with methods that use other components from the context.

Here are some example of rule creating methods found in the implementation (pseudocode):

```
//make a rule to travel between locations
Rule GoToRule(Location from_scenario, Location to_scenario)
{
      Rule rule;
      rule.precontitions.Add(Label("to_scenario NOT locked"));
      rule.precontitions.Add(Label("Player is in
from_scenario"));
      rule.postconditions.Add(Label("Player is NOT in
from_scenario));
      rule.postconditions.Add(Label("Player is in to_scenario"));
      return rule;
}


//make a rule to pick an item from the ground
//(each Item knows its location, type, quantity…)
Rule PickItemRule(Item item)
{
      Rule rule;
      rule.precontitions.Add(Label("Player is in
item.location"));
      rule.precontitions.Add(Label("item is dropped on the
ground"));
      rule.postconditions.Add(Label("item is NOT dropped on the
ground));
      rule.postconditions.Add(Label("Player has item"));
      return rule;
```

```
}
```

Method parameters could be randomly chosen from the rest of the context.

Also, rules are added in packs to get particular properties. For instance, we wanted to make possible to travel from on scenario to another always both ways, so we always add movement rules by using:

```
//fully connect two locations
//using the GoToRule method decribed before
Rule[] ConnectLocations(Location scenario_A, Location
scenario_B)
{
     Rule[2] rules;
     rules[0] = GoToRule(scenario_A, scenario_B);
     rules[1] = GoToRule(scenario_B, scenario_A);
     return rule;
}
```

This can be used to produce complex plot scene parts, using *Information* labels, as in the following example set of rules;

Preconditions	**Rule**	**Postconditions**
Player is in the same location than the little boy Player **does not** know that the dragon killed little boy's father	A llitle boy explains the player a dragon killed his father, and asks the player to seek revenge for his father by killing the dragon.	Player **knows** that the dragon killed little boy's father
Player **knows** that the dragon killed little boy's father Player is in the same location than the dragon The dragon is alive	The player fights the dragon and (hopefully) wins.	The dragon is dead.
Player is in the same location than the little boy The dragon is dead. Player **knows** that the dragon killed little boy's father	Player talks with the boy, who gives the player his father's helmet as a reward.	Player has father's helmet.

Table 1: chained plot sequence with rules example

It is important to notice that by using the information label "Player knows that the dragon killed little boy's father" and its negative the three rules are linked and have a particular order. This way, when the quest building algorithm tries to introduce one of the three rules into the quest, all three will forcefully be introduced in the same order, making this way a larger story arc. Very complex story arcs can be built up by using this technique, even with random variations that swap rule groups by other ones inside larger chained rule groups.

6 - The Game

The system produces quests to be solved by players in a game, who interact with scenarios, characters, enemies, etc. that exist in the game world. This section is divided in two main parts. In the first part, we will see how the properties of a game would affect the rest of the system. In the second part we present our particular game, and how this particular game and the rest of the system are linked.

a. Impact of the game properties in the system

I - Eligibility of the game

The whole system is aimed to produce a linked series of actions that change the game state. The player is meant to perform the actions in a particular order (or some acceptable variations of that order, see section 8, last paragraph), and that sequence is supposed to make the game state evolve so as to contain certain goal properties.

Considering this, not all the type of games are eligible for applying procedural quest generation: not every videogame supports containing a linked sequence of actions, or even a rich enough game state to play with. We could, as an exercise, try to apply the logic of the system in existing games, imagine the result and evaluate if generating procedural quests in this game brings some value.

As an example, arcade games will probably fail this test because the game state is reduced and highly volatile (every gameplay resets the whole state, no data is preserved). Lots of action type games would fail too, because the set of possible actions tends to be small, and the order these actions is decided by players by using reflexes and dexterity, so a pre-established order is not desirable. On the other hand, most graphical adventure and role playing games would take profit of procedural generated quests, as would any game that features a rich enough open world.

Summing up, the system will only be desirable for games that feature a large enough set of actions that could be linked, a rich game state, and a context that would take benefit of structuring the player goals as quests.

II - Semantic

The semantic of a game is the conceptual plane in which labels, game objects and rules exist and their meaning. The semantic of a game is the set of ideas the game presents to the player and uses internally as game mechanics.

The game's semantic defines the types of labels and rules that will be used in our system. In the label *"Player has two coins"* we can notice that the game semantic is what gives a sense to "*has"* and "*coins"*: "*has"* means that there is something stored in the player's inventory (or backpack, or pocket…). For the system, "*has"* is only a type, a property, but the semantics will determine how this label will be operated with according to the concept it represents (see last paragraphs of section 5.b for some examples of meaning influencing label operations). Moreover, the videogame will recognize what has to be done with a label by this type: if the label means the player *has* something, the game program will use the information that label bears to fill the inventory. The term "*coin"* defines the kind of item: inside the label, it may be just and identifier, but for the game is a particular idea, possibly related to a name, an image, a game graphic, a game bonus… This could have some kind of impact on the rest of the system, as we explore in the next section.

Summing up, semantic will determine the objects and their properties, labels and rules the system will need to build procedural quests, and the meaning of them. The meaning of labels will define label operations.

Example: **Pong**

"Pong is a two-dimensional game that simulates table tennis. The player controls an in-game paddle by moving it vertically across the left side of the screen, and can compete against either a computer controlled opponent or another player controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and forth. The aim is for a player to earn more points than the opponent; points are earned when one fails to return the ball to the other." [9]

Pong is considered the "first videogame" in computer history. We have used this very simple game to identify some of the concepts mentioned in this section.

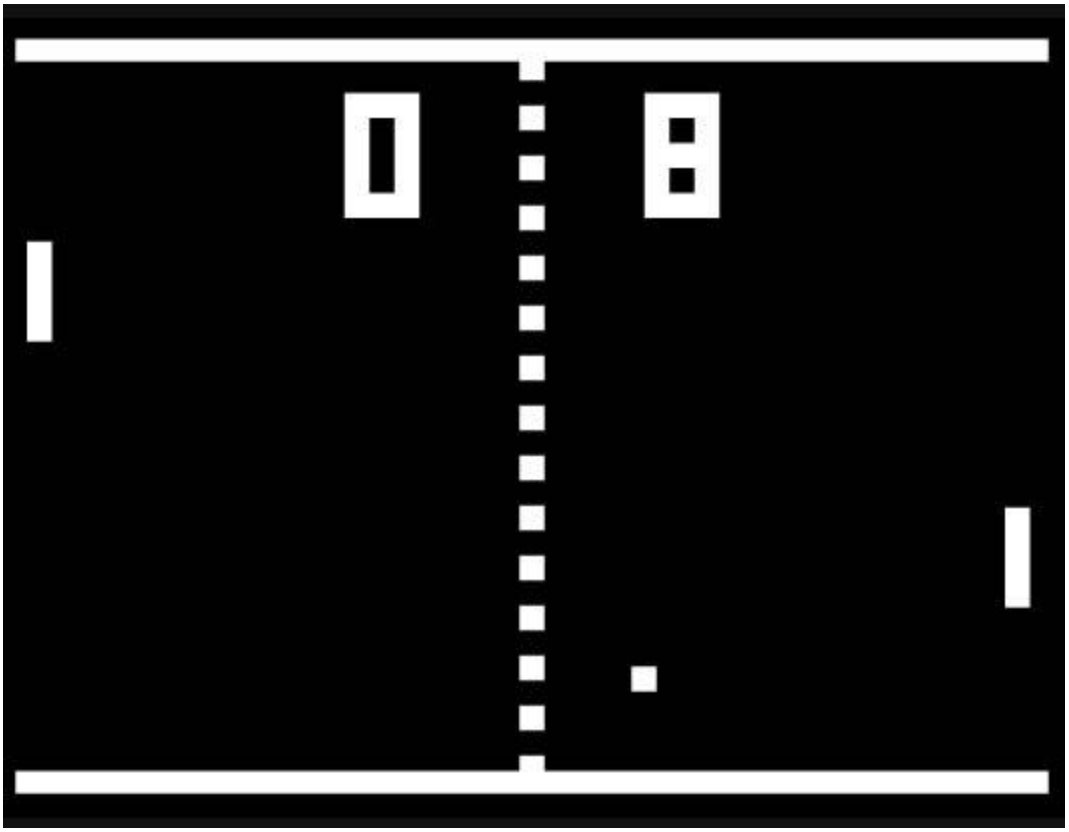

Image 1: Screenshot of Pong

Game objects in Pong are the paddles, the ball, the walls and the field.

The game state features the position of the bars, position of the ball, ball vertical and horizontal velocity, and player scores. The set of labels would be:

[paddle1_position, X], [paddle2_position, X], [ball_position, X, Y], [ball_velocity, X, Y], [player1_score, X], [player2_score, X]

Among the rules can be found: "a bar hits the ball", "the ball moves", "the ball scores", "a bar moves" and "a wall hits the ball".

For instance, "hits the ball" rule would have, as precondition, the ball being placed correctly according to the bar so the hit can be produced, and a postcondition would erase the previous ball velocity and set a new one with the X property inverted. Another example, "the ball scores" rule would have as precondition the ball having surpassed a player's score line coordinate, and as postcondition, it would update the player score and reset the position and velocity of the ball.

III - Logic of the game world

Every game possesses a certain logic bound to the fictional logic of its world. Usually, that logic resembles real world's logic, with some flexibility: for instance, a

person can only be in one place, it is not possible to travel in time, dead people cannot speak… but in a game that logic may be different, and characters can be in many places at the same time, travel to the past, or become ghosts and talk with alive characters. In games believability is not a key issue compared with consistency: travelling in time is not believable (so far), but if you assume it as being possible the story and possibilities of the game must be consistent with that.

This influences the system not only by determining the conditions handled by rules, as we saw in the previous section, but also, and more importantly, by conditioning the population of the game context and rule initial set.

When initializing the game, the context need to be populated with instances of the different objects handled by the game (e.g. characters, items …), an initial set of labels (e.g. “Jack is inside the tavern”, “Jack has 2 coins” …) and an initial set of rules. If the game world logic permits a character to be in two different places, the context populator could add two times a label that places a character in a place, for two different places, if the existence of a character implies it could be killed and it could be spoken to, then the populator has to add a rule to kill the character and a rule to speak to the character every time a character is added to the context, and so on so as to fully represent how the game works when the initial context is being created.

b. Our game

For this project, a simple game has been developed so as the have a fully functional case to work with. It could be described as the base of a classic role playing game, a functional skeleton that features the base mechanic that almost every role playing game (RPG) features. This particular genre is where the concept of *quest*, appeared and has become a defining trait of the genre.

In this game the player controls a knight through a third person camera. The knight can be moved using the usual control axis (key arrows, WASD keys, joystick….) or by clicking somewhere in the scenario. The knight can swing its sword to fight enemies (Q key), and talk to characters (E key).

Image 2: right, game main character, who represents the player and left two game non-player characters (NPC)

Players can travel to eight different locations so far: clicking the compass button on the head up display (HUD) makes a portal screen show up, displaying the different locations connected to the current one. The player may go to one of the connected locations (displayed with a blue line) by clicking on its icon, as long as the way between the locations is not blocked (displayed with a red line). Blocked ways may be opened with in game procedural rules.

Image 3: Scenario selection and travel screen

Image 4: Game scenarios, left to right, up to down: Cemetary, Beach, Volcano, Desert, selection screen, Tavern, Fields, Castle and Cave

The game may contain up to ten different characters, depending on how many characters the context populator decided to insert in the game. There are eight types of items that may be held by characters, the player, and enemy group or just dropped in the ground of the scenario, ready to be picked up. Both characters and items can be easily be added progressively.

Image 5: Game characters (models borrowed from the game "The Legend of Zelda: Twilight Princess", then added custom rig and animations)

Image 6: Game items (including coins, sword, grail, key, crown)

Enemies are organized as enemy groups. Each enemy group is defined by a difficulty level, the scenario where it spawns, an item it may drop, and its current state. The level indicates which enemies, and which number of each type the group contains.

Image 7: Game enemies (models borrowed from various sources, added custom rig and animations). Also displayed quest list interface at the right, and quest part details bottom.

The player may kill the enemies by hitting them with the combat move. When each enemy is hit, its health is lowered. When an enemy health is fully depleted the enemy disappears. Health of enemies is shown by the red marker their wear. When all the enemies of the group have been killed, the group is considered to be vanquished, and the item they might keep is dropped to the ground. Currently, the game features three types of enemies.

Image 8: Enemy health marker display, left is maximum health, left is half health

The player may start dialogues with characters, which can trigger events or imply item trading, information retrieval …

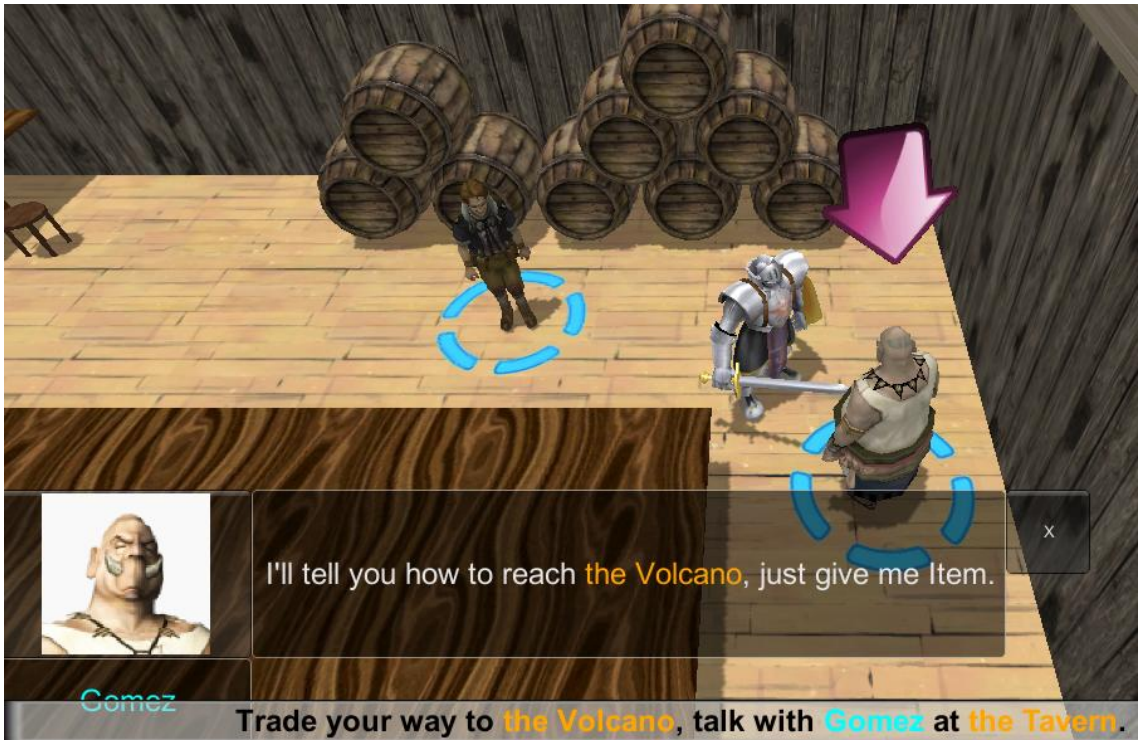

Image 9: Dialogue with NPC inside the Tavern

The game also handles *informations*, an abstract kind of object that provides means for dialogues to influence on quests, or force a particular order between different rules (see table in section 5.c for an example).

Another type of object is on development and scheduled to be featured in the game: scenario active objects. This objects of this kind are scenario-bound objects that appear if introduced to the context, and the player may interact with them to trigger particular rules. These items could be objects like statues, totems, portals, magical trees…

The whole game has been developed with the game engine Unity 3D, and many graphical assets from other existing games have been used to speed up the game development.

Game integration

The way the game and the rest of the system interact is a critical part of the development. In this particular case, we decided to deeply link both sides: the game world builds itself by reading the system context. When a scenario is entered, the game program reads the whole current context looking for objects (characters, enemies, items…) that are present inside that scenario, than instantiates them. The program also looks for properties that objects have in the current set of labels.

This technique makes the game world highly consistent with the abstract model of itself (the context), and is updated as rules are being applied. The player actions trigger rules from the context, then only after each rule is applied to the context, the game world is updated as explained before.

Location linking is achieved by introducing movement rules: two scenarios are linked because the context contains a rule that makes possible to travel from one to the other and vice versa. Other game logic could allow one-way connections between locations, but for this particular case we preferred to always link scenarios both ways.

Interface representation of the quest, and the game texts are also build up by reading information from the abstract instance of the game.

All this makes the game world a visual representation of the abstract objects handled by the system. As a desirable property, this method brings is the modularity of the game: changing the visual assets, or the whole game engine would preserve the game intact. Thus, the game could be exported to be represented elsewhere (a web page, a command console, or another completely different visual engine), but the abstraction of the game and the rest of the system would remain the same and still be functional.

So far, the game's set of labels is:

Label	Meaning	Parameter 1	Parameter 2
Has	Player has item type	Item type ID	Quantity
Is	Player is in scenario	Scenario ID	
Talked	Player talked to character	Character ID	
Explored	Player explored location	Scenario ID	
Knows	Player know some piece of information	Information ID	
Killed	Player killed enemy group	Enemy group ID	
Locked	Scenario is locked	Scenario ID	
DroppedItem	Item is dropped in scenario	Item ID	
Alive	Character is alive	Character ID	

Table 2: Example labels

Some example of rules the game system features:

Rule	Meaning	Preconditions	Postcondition
GoTo	Player travels between scenarios	¬Locked(scenarioA) ¬Locked(scenarioB) Is(scenarioA)	¬Is(scenarioA) Is(scenarioB)
Reach	Player reaches scenario	¬Locked(scenarioA)	Is(scenarioA)
TalkTo	Player talks to character	Alive(character) Is(character.scenario)	Talked(character)
Pick	Player picks item	DroppedItem(item) Is(item.scenario)	¬DroppedItem(item) Has(item.id, item.quantity)
Trade	Player trades item per item	Has(itemA, quantity)	¬ Has(itemA, quantity) Has(itemB, quantity)
CombatDrop	Enemy drops item to the ground	Killed(enemy)	DroppedItem(item)
Kill	Player kills enemy group	¬Killed(enemy)	Killed(enemy)
AskCombatHelp	Character asks for help to fight	…	…
Reward	Character gives item to player as a reward	…	…
…	…	…	…

Table 3: Example rules

7 - Procedural quest generation

In this section we will explain the process of making a single quest, given a particular context. Optimal quest searching in section 8.

The procedural quest construction is quite complex algorithm. It is fully **determinist**: if it runs on the same context twice, the result will be the same. The search process needs it to be determinist: that way the search process is able to modify the context and compare the quest it produces to the quest other contexts produce. If the same context may produce different quests, the search process could not use the context as the variable. Retrieving terminology from other common algorithms, the context here is the **seed** of the process

The algorithm needs one more thing besides the context to start: **the target**. The target is a label that we look for along the whole quest, the main goal. It can be any kind of label: if we want the player to find the Holy Grail, the target will be "Player has Holy Grail", if we want the player to kill the dragon, "Dragon is not alive", etc. There is no problem if the target is accidentally among the initial set of labels, or it is too easy to achieve or just impossible, the search process will handle that kind of issues. The target could be more than one label with some minor trivial changes in the algorithm, for this project we will use a single label as target.

The quest is built backwards, using the target as starting point and trying to reach the initial state by stacking rules that will have to be applied in the inverse order. As a consequence, all resulting quests are forcefully **valid**. If the quest would be impossible to achieve, the construction algorithm fails.

The algorithm handles two critical label lists, and a rule list. The rule list is the **produced quest**: it is an ordered list of the actions the player will have to do so as to complete the quest. The label lists are:

- **Pending** labels: these are labels that need to be at some point in the context label list (the game state). The algorithm will look among the rules postconditions for pending labels: the rules that have those labels in their postconditions are rules that introduce the labels in the game state, so they are desirable to enter the quest list. The pending labels list is initialized with the **target**.
- **Needed** labels: needed labels are labels that rules added to the quest need to be applied. Rules that are added after others cannot produce conflicts with labels present in the needed labels. For example: if the player needs to talk to the tomb robber to trade the Holy Grail, it cannot happen that the robber is killed by the dragon before the player trades with him. There is no problem if the dragon wants the tomb robber as lunch after that, though. As the quest is built backwards, this list is needed to keep logical consistency through the quest and avoid asking the player to do impossible things (like talking with a

tomb robber that has been eaten by a dragon). A last important thing to consider about the needed label list: if a rule that produces a label that is present in the needed label list is introduced in the quest, then that label may be removed from the needed label list. Keeping with the same example: if the player, before talking with the tomb robber, has to talk with a voodoo shaman that brings dead characters back to life, then there is no problem if the dragon kills the tomb robber. Note this list could be stored at each step and used as fail condition in the game.

During the process, every time a new rule for the quest if being searched, the context rule list is gone through two times:

The first time the algorithm looks for rules that produce no conflicts and that have postcondition labels which can be found in the pending labels list too. Rules chosen this way help directly to solve the quest.

The second time through, any rule (as long as it produces no conflict) can be picked, even if it doesn't seem to help the quest to be solved. This second pass helps the process keep running by adding rules that change the context and may turn possible to add in the next round a rule that couldn't be picked before because it produced conflict.

If a rule that produces no conflict cannot be found during the second pass, the algorithm has failed to find a suitable rule, and the quest is considered to be impossible to be resolved with the provided context.

The process may be refined to get better results. For instance, in our implementation we added a third pass when looking for a suitable rule. This third pass is introduced after the two previously explained ones, and in the second pass we ignore some types of rules. In the first look through we ignore movement rules to avoid making the player wander meaningless through the map as much as possible: movement rules are almost always possible to be added, but do not bring much value nor change to the game state, so other kinds of rules are tried before movement rules.

The algorithm flow is:

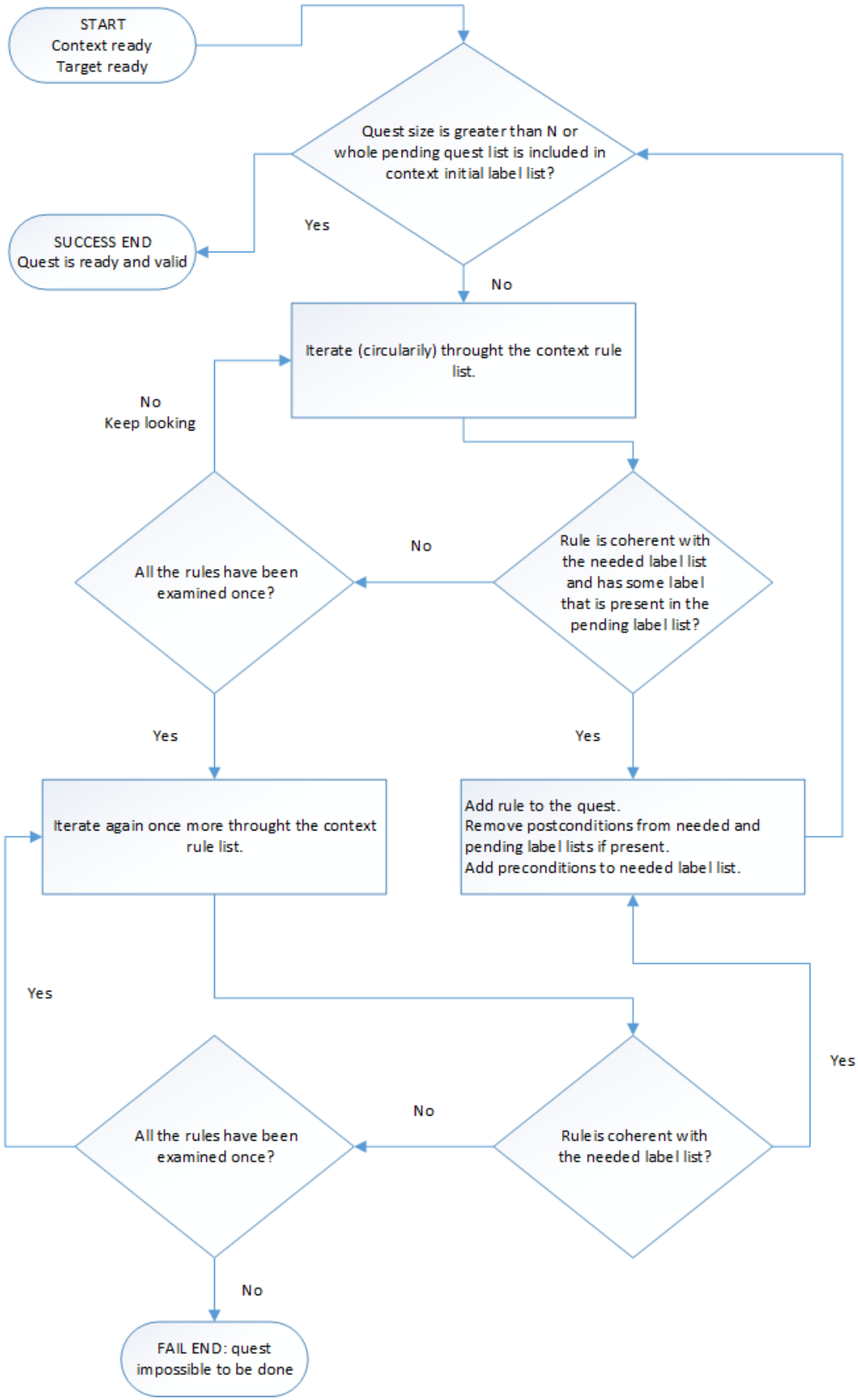

Figure 2: Quest generation algorithm flow

In our implementation of the generator, produced quests pass a final filter that group some kinds of rules as a single one. This filter aims to make the quest more user-friendly, as some kinds of rules or rule combinations are too restrictive. For example, movement rules when stacked by the quest generator only takes into account one possible way: ("*Go to the castle, then go through the desert and finally enter the tavern*"), but maybe other valid possibility to reach the tavern exists (like "*Go to the castle, then cross the river, and finally enter the tavern*"). The filter would replace movement rules by a more generalist one, like "*Reach the tavern*". After this modification, the quest is still logically valid, as the generation algorithm has verified it is possible to reach the tavern. The player may now reach the tavern going through the path the generation algorithm considered, but also through other possible paths the player may discover.

8 - Quest evaluation and search algorithm

The search algorithm is based on the Variable Neighborhood Search (VNS) and Simulated Annealing (SA) metaheuristics.

Pierre Hansen and Nenad Mladenovi'c, who formalized VNS in 1997 [10], described VNS as a *metaheuristic for solving combinatorial and global optimization problems whose basic idea is systematic change of neighborhood within a local search*. Paraphrasing this description, VNS is a method for finding optimal solutions which explores the set of possible solutions by iteratively evaluating close possibilities.

In this project, a context is related to one and only one quest. For this implementation of VNS, the algorithm will explore the neighborhood of contexts, taking a random one as starting point, and exploring close ones iteratively. Nevertheless, quests, instead of contexts, are evaluated so as to find the best solution. As VNS implies, best quest found so far is compared to a quest produced with a slightly modified context, and the better of the two is kept as best solution. The process repeats for a user determined number of times.

Simulated Annealing (SA) is applied when producing neighbor contexts of the one kept by VNS each round. The project contains a library of context modifiers, little methods that produce a particular, randomized change in a given context. For instance:

- Swap the order of two rules
- Add a character
- Change the location of an item
- Add a movement rule
- Add a label that blocks a scenario
- Etc.

SA decides which modifier will be applied every time a context is being altered to get a neighbor solutions for VNS. Simulated Annealing is a probabilistic Monte Carlo method applied to find extremes of a large universe of possible situations [11] [12]. For this project, SA is not applied to find the optimal quest: as mentioned earlier, SA decides which modifier is applied on a context. Each modifier possesses a score, and after a context is modified and evaluated, if the result is better than the previous one, the applied modifier is scored positively, otherwise its score is decreased.

The SA framework, during the first iterations of the search algorithm, experiments all kinds of modifiers: at first their scoring is not very significant, so any modifier can be chosen. As the iteration number gets closer to the maximum iteration limit, SA will progressively choose modifiers that produce better results, making the process more selective.

At the end of the whole process, the quest kept by VNS is passed to the game, then loaded to be played.

Quest evaluation depends on the nature of the game, and the game designer's criteria. Some possible criteria are:

- Length of the quest
- Variety of rules involved
- Balance of the game rhythm
- Number of scenarios visited
- Number of character met
- Difficulty
- Inclusion of a certain type of rules (dialogue, combat…)
- Etc.

In the next section we explain how to use the evaluation function as the point to introduce the interactivity with the player.

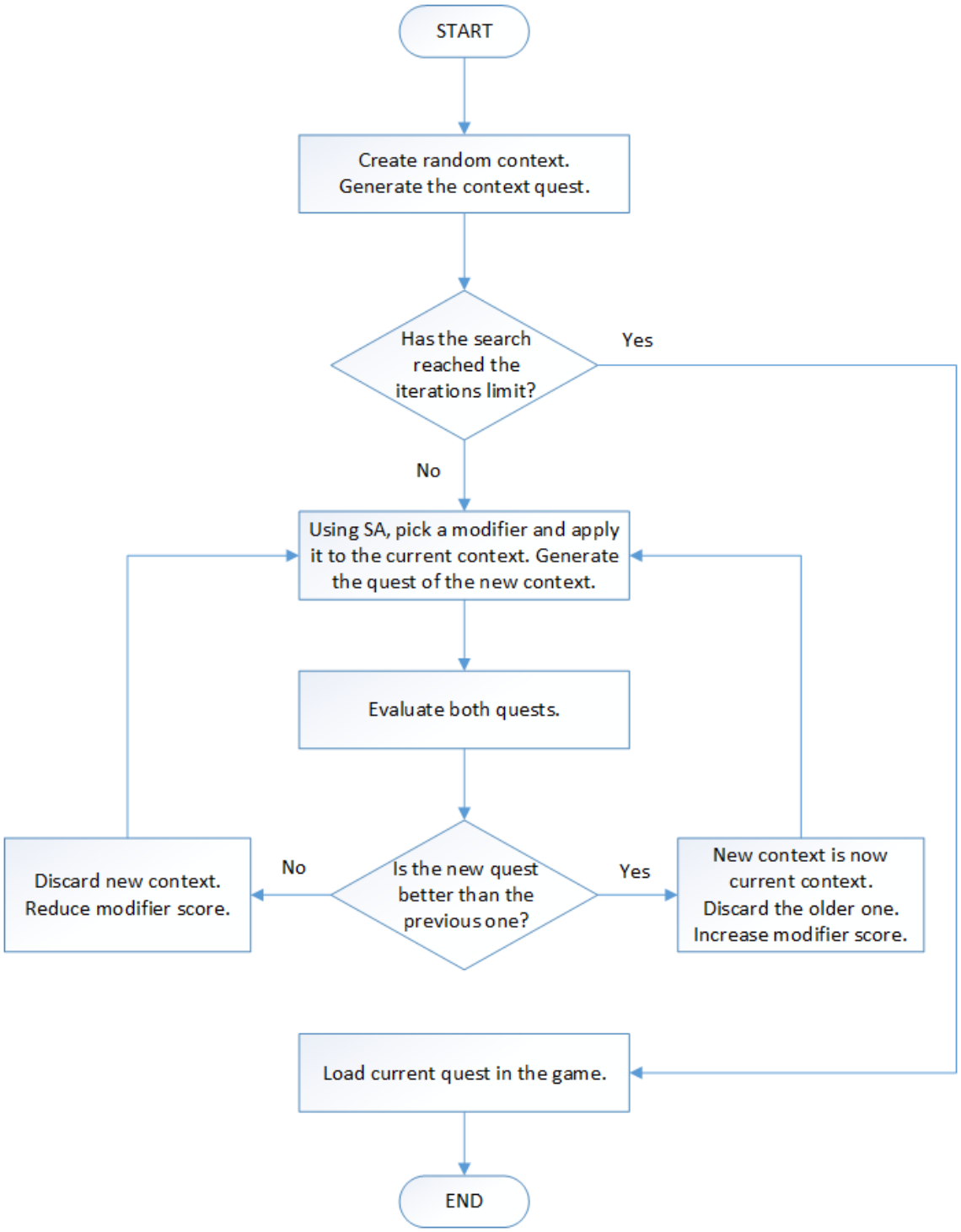

Figure 3: Search algorithm flow

9 - Interactivity and system feedback

Interactivity of the process is achieved by influencing the quest evaluation function criteria. For instance, the system could use the evaluation function developed by D. Thue and V. Butliko in PaSSAGE [3]: given a player profile, the quest score is calculated by its matching level with the profile. Other approach could be having the player fulfill a survey before playing, then the quest would be evaluated with the information delivered by the player.

An interesting approach is to collect information from the player implicitly during gameplay, and use the gathered data to influence the quest evaluation. First, the player would play some non-adapted quests that could be specifically done to gather a considerable amount of initial data, then progressively generate more adapted quests. Information of the player preferences could be gathered by introducing choices (e.g.: make the player choose between a fight or a dialogue), registering in which order the player confronts quest parts, or the time spent doing them. This feedback is then transformed into evaluation criteria by ranking rule types then scoring quests using that ranking.

As the evaluation function scores higher quests that match better to the player, the process becomes interactive, as the search process will produce player-adapted quests.

Although the project includes the structure to support the interactivity of the quest generation process, this feature is not operational in the present version of the game. The full development of an interactive feedback process from the user is one of the open research activities of this project.

10 - Results

Fully developing and implementing the system has brought the possibility of facing each and every aspect of it. The practical implementation has revealed all the problems and flaws of the design, making possible to correct and improve the design. As result, the system is ready to be exported to other practical cases.

In fact, we consider the system could be suitable as the base or extension of a usual, commercial videogame: with a large enough rule, label and game object libraries, a full game can be developed.

Many improvements of the system were unexpected. For instance, the following parts were not planned in the first drafts of the system:

- Methods to generate rules and chained rules collections (these methods provide parameterized templates to produce rules according to the game needs).
- Label operation differences defined by label type: depending on the game needs, labels need to be added following different definitions.

Also, in the first place a quest validator was planned, but the algorithm developed guarantees the validity of the generated quests. No impossible mission is delivered to the game.

Due to the small dimensions of the rule and label library implemented so far, quests show a lack of originality. Nevertheless, the game is still playable and entertaining, and the generated quests already resemble those found in well-known role playing games.

Properties of the system also hint promising applications in the field of plot construction. If rules were plot segments, the system could be easily arranged to build logically linked stories, and the search algorithm would provide these stories with rhythm, plot twists and other desirable properties.

Still, if we limit our sight to videogames, the system has shown to be able to build quality content for games without human supervision. The quality and variety of the content is improved, as said, by expanding the system libraries, letting the core functionality of the system unchanged.

Quest search can be enhanced working further on the evaluation criteria and expanding the context modifier battery. The search algorithm, though, would remain the same.

The main benefit the system has proven to bring is the game design efforts multiplications, as it magnifies and endlessly extends the designer work, which for a game using the system is designing rule, game object and label libraries, rule creation battery and quality criteria. The system uses the designer creation to generate an infinite collection of quests, providing this way and endless resource of content for the game.

Bibliography

[1] Intel Corporation. (2012, May) [Online]. Available: http://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html

[2] J. Togelius, E. Kastbjerg, D. Schedl, Yannakakis, "What is Procedural Content Generation? Mario on the borderline.", IT University of Copenhaguen, Copenhaguen , Denmark (2011).

[3] D. Thue, V. Butliko, "PaSSAGE: Past, Present, and the Road Ahead" University of Alberta, Google. Mountain View, CA. (2009, August 3)

[4] F. Peinado, "Interactive digital storytelling: Automatic direction of virtual environments", Universidad Complutense de Madrid, Madrid, Spain (2006)

[5] M. Tomin, J. Liu, "Interactive Storytelling in Entertaining", (2008)

[6] R. D. Laws. (2002) Robin's Laws of Good Game Mastering. Steve Jackson Games, first edition.

[7] F. Peinado, P.Gervás, "Transferring Game Mastering Laws to Interactive Digital Storytelling", Universidad Complutense de Madrid, Madrid, Spain (2004)

[8] M.Riedl, D.Thue, V. Butliko, "Game AI as Storytelling", University of Alberta, Edmonton, Alberta, Canada (2011)

[9] Wikipedia contributors. (2013, December 22). Pong. [Online]. Available: http://en.wikipedia.org/wiki/Pong

[10] Nenad Mladenovi´c, Pierre Hansen (1997). "Variable neighborhood search". *Computers and Operations Research* 24 (11): 1097–1100.

[11] Kirkpatrick, S.; Gelatt Jr, C. D.; Vecchi, M. P. (1983). "Optimization by Simulated Annealing". *Science* 220 (4598): 671–680.

[12] Černý, V. (1985). "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm". *Journal of Optimization Theory and Applications* 45: 41–51.

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Wed Jan 08 17:11:18 CET 2014
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)